Teaching Geometric Algebra with CLU$Calc$ (Innovative Teaching of Mathematics with Geometric Algebra)

AUTHOR(S):

Perwass, Christian

CITATION:

Perwass, Christian. Teaching Geometric Algebra with CLU$Calc$ (Innovative Teaching of Mathematics with Geometric Algebra). 数理解析研究所講究録 2004, 1378: 33-50

ISSUE DATE:

2004-05

URL:

http://hdl.handle.net/2433/25644

数理解析研究所講究録 1378 巻 2004 年 33-50

Teaching Geometric Algebra with CLU*Calc*

University of Kiel, Cognitive Systems Group
Christian Perwass

Abstract

This text gives a short overview of the features of the visualization software CLU*Calc*, and how this software may be used to teach various aspects of Geometric Algebra. CLU*Calc* is a stand-alone software tool that uses the 3d-graphics library OpenGL to visualize the geometric meaning of elements of Geometric Algebras. These visualizations can very easily be animated or made user-interactive. Apart from this, CLU*Calc* offers many more features like the annotation of graphics with LaTeX text, the visualization of circle-valued functions, drawing of transparent objects, user defined lighting, error propagation and much more.

1 Introduction

Geometric Algebra (GA) has found its way into many areas of science, since David Hestenes treated the subject in the '60s. Hestenes' aim was in particular to find a unified language for mathematics, and he went about to show the advantages that could be gained by using GA in many areas of Physics and geometry [HS84, HZ91]. Many other researchers followed and showed that applying GA in their field of research can be advantageous, see e.g. [LFLD98, PL01, Dor01, LHR01].

However, since GA is usually not taught in high school or at universities, the subject is not well known and seems mostly to be regarded as a very specialized and very difficult mathematical tool. This is the case even though many features of the algebra have a direct geometric interpretation and should thus lend themselves to simple explanations. One reason for this discrepancy may be that it is not always easy to imagine the geometric relations, and for a teacher it is not easy to draw them.

In order to remedy this situation, Leo Dorst had developed a MatLab toolbox called GABLE, which allowed the user to visualize GA elements in 3d-Euclidean space [DMB00]. Unfortunately, elements in projective or conformal space could not be visualized. I therefore started in 2001 to develop a C++ software library, which could automatically interpret elements of a GA in terms of their geometric representation and visualize them. This visualization was done using the OpenGL 3d-graphics library and worked for 3d-Euclidean

space and the corresponding projective and conformal spaces. The software library, called CLU*Draw*, was made available in August 2001 under the GNU Public License agreement as OpenSource software.

The drawback of this library was, that a user had to know how to program C++ in order to use it. Furthermore, every time the program was changed, it had to be re-compiled in order to visualize the new result. Many people that might have been interested GA could therefore not use this tool, and even for those who did know how to program in C++, it was quite tedious to constantly re-compile a program when a small change was made to a variable.

I therefore decided to develop a stand-alone program, with an integrated parser, such that formulas could be typed in and visualized right away. This software, called CLU *Calc*, was first made available for download in February 2002. As of November 2003 it is available in version 3.0, including many advanced features like solving simple multivector equations, error propagation calculations, text annotations using LaTeX, and support for a presentation mode, which integrates interactive and animated 3d-graphics in presentation slides.

The basic philosophy behind CLU*Calc* is that the user has a GA formula and would like to know what this formula implies geometrically. I therefore developed the parser to be used in CLU*Calc* from scratch. In this way I could choose the symbols representing the different products in GA such that they are as close as possible to the ones used on paper. The resulting script language is called CLU*Script* [Per03]. For example, the formula

$$Y = (A \wedge B) \cdot X \quad \text{becomes} \quad \texttt{Y = (A ^ B) . X}$$

when written in CLU*Script*. In order to visualize the result of this calculation, CLU*Script* defines the colon-operator (:), such that `:Y = (A ^ B) . X` visualizes `Y`.

In this text I will try to show the potential CLU*Calc* offers for teaching GA, by demonstrating different aspects of GA. I will also demonstrate features of CLU*Calc* which have no direct relation to GA. This is to show that CLU*Calc* may also be used to teach other subjects than GA. In the following I will assume that the reader is familiar with the basic concepts of GA. An introduction to the fundamental aspects of GA can be found in [PH03], which includes an "interactive" introduction using CLU*Script*. Mathematically rigorous and more in-depth introductions and discussions of GA (aka Clifford algebra) can be found, for example, in [HS84, Hes86, HZ91, GLD93, Lou97, Rie93, GM91, Por95, LFLD98, Dor01, Per00, DMB00]. The example scripts presented in the following will run with CLU*Calc* 3.1, which may be downloaded from `www.clucalc.info`.

2 Using CLU*Calc*

In order to describe geometry, typically the Geometric Algebra of 3d-Euclidean space or the corresponding projective and conformal spaces are considered. In each space vectors

and blades usually represent different geometric entities and therefore have to be analyzed differently. CLU*Calc* allows the user to work in all three spaces mentioned above concurrently and to transfer vectors from any one space to any other. Here is an example script:

```
?Ae = VecE3(1,2,0); // Create vector in E3 at position (1,2,0)
?Ap = VecP3(Ae);    // Embed vector Ae in projective space
?An = VecN3(Ap);    // Embed projective vector in conformal space
```

The question mark at the beginning of each line is an operator that prints the contents of the element on its right in a text window. The output of the above script is

```
Ae = 1^e1 + 2^e2
Ap = 1^e1 + 2^e2 + 1^e4
An = 1^e1 + 2^e2 + 2.5^e + 1^e0
```

In projective space the homogeneous dimension is denoted by **e4**. In conformal space **e** denotes e_∞ and **e0** denotes e_o. Moving between the different spaces is particularly useful to show the different geometric interpretations of blades in the different space. However, note that blades cannot directly be transferred from one space to another. This has to be done via the constituent vectors. Here is an example:

```
Ae = VecE3(1,0,0);    // Create unit vector along x-direction
Be = VecE3(0,1,0);    // Create unit vector along y-direction

:Red;      // Switch current color to red.
:Ae^Be;    // Visualize the outer product of Ae and Be

:Blue;                  // Switch current color to blue.
:VecP3(Ae)^VecP3(Be);   // Visualize the outer product of Ae and Be
                        // when embedded in projective space

:Green;                 // Switch current color to green.
:VecN3(Ae)^VecN3(Be);   // Visualize the outer product of Ae and Be
                        // when embedded in conformal space
```

Note that the colon is an operator that tries to visualize the element on its right. The LHS of figure 1 shows the resultant visualization of this script. The disc is the result of `Ae^Be`, the line the result of `VecP3(Ae)^VecP3(Be)` and the two points are the result of `VecN3(Ae)^VecN3(Be)`. The disc represents the subspace spanned by **Ae** and **Be**, whereby the area of the disc is the magnitude of the blade `Ae^Be`. In projective space the outer product of the embedded points `VecP3(Ae)` and `VecP3(Be)` represents the line passing through the points **Ae** and **Be** in Euclidean space. Hence, the representation as line. The

length of the line gives the magnitude of the corresponding blade. The outer product of the two vectors when embedded in conformal space (`VecN3(Ae)^VecN3(Be)`), represents the point pair (`Ae`, `Be`) in Euclidean space and is visualized accordingly. Note that such a point pair is in fact a one-dimensional sphere.

The RHS of figure 1 shows the same for the outer product of three vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$, when regarded as Euclidean vectors and embedded in the corresponding projective and conformal space. The outer product of three Euclidean vectors in 3d-Euclidean space, represents the whole space. This is represented by the cube. The volume of the cube is equal to the magnitude of the blade. When these three vectors are embedded in projective space, their outer product represents a plane in Euclidean space, which is represented as a disc. Again, the area of the disc is the magnitude of the blade. The outer product of the three vectors when embedded in conformal space represents a circle through these three points in Euclidean space. Hence, the circle drawn through the three points. The radius of the circle can be extracted from the blade, but is not directly related to the magnitude of the blade in conformal space.

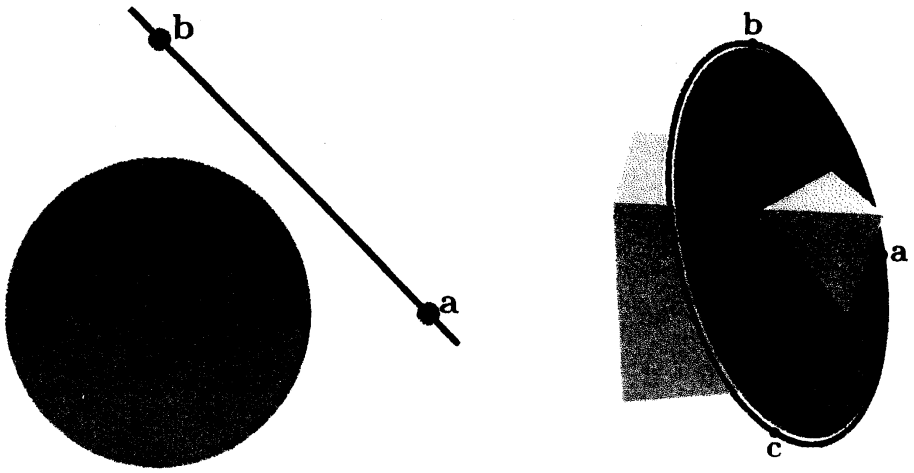

Figure 1: The left image shows the geometric interpretation of the outer product of two Euclidean vectors (the disc) and the geometric interpretation of their outer product when the vectors are embedded in the corresponding projective (the line) and conformal (the point pair) space. The right image shows the same for three vectors. The Euclidean, projective and conformal interpretations are now the whole space, the disc and the circle, respectively.

Note that the script shown above does not automatically generate the annotations seen in figure 1. This has to be done with the help of additional commands as will be explained later on. Nonetheless, the images in figure 1 were generated directly with CLU*Calc*, and for the annotations it is possible to use arbitrary LaTeX code.

Intersections of geometric entities can also be evaluated and visualized quite easily. In GA the *meet* of two blades gives the blade representing the largest subspace the two blades have in common. Geometrically speaking this is an intersection. In CLU*Calc* the operator & is used to evaluate the meet of two multivectors. Note that when & is applied to two integer values, it evaluates their bit-wise AND. In the following example script two spheres represented in conformal space are intersected and the resulting circle is intersected again with a plane.

```
SetPointSize(6); // Increase size of points for better visibility
SetLineWidth(4); // Increase width of lines for the same reason

DefVarsN3(); // Define standard variables for conformal space
:N3_SOLID; // Display spheres as solid objects
:DRAW_POINT_AS_SPHERE; // Draw points as small spheres and not as dots

//      red   green  blue   alpha = transparency
:Color(1.000, 0.378, 0.378, 0.8); // Use a transparent color
:S1 = SphereN3(VecE3(1), 1); // Make center of first sphere user-interactive

:MBlue; // Medium blue
:S2 = SphereN3(0.5,0,0, 1); // Another sphere of radius 1

:Green;
:C = S1 & S2; // The intersection of the two spheres

:Orange;
// Create plane, whereby 'e' is a predefined variable denoting
// the point at infinity. 'e' is defined through the function
// DefVarsN3() which was called in the fourth line.
// The scalar factor only increases the size of the plane visualization.
:P = 10*VecN3(0,0,0) ^ VecN3(1,0,0) ^ VecN3(0,0,1) ^ e;

:Magenta;
:X = P & C; // Evaluate intersection of circle and plane
```

Figure 2 shows the result of the above script. Note that all intersections where evaluated with the same operation: the meet. The meet operator, as implemented in CLU*Calc*, regards only the algebraic properties of the multivectors and does know nothing about their geometric interpretation. It is, in fact, a direct implementation of the mathematical definition of the meet. The center of the sphere S1 in the above script, can be changed by the user interactively using the mouse. This will be explained in more detail in the next section. If S1 is moved by the user, such that the spheres do not actually intersect anymore, the meet operation returns an algebraic object that can be interpreted as a circle with imaginary radius. This is also displayed by CLU*Calc* as a dotted circle. Imaginary

spheres and point pairs exist as well and are also visualized as transparent spheres and two points connected with a dotted line, respectively.

Typically one is also interested in visualizing functions. With CLU*Calc* scalar valued, vector valued and certain multivector valued functions can be drawn. One particularly interesting feature is the visualization of circle-valued functions. Since a circle can be represented in conformal space as a trivector, i.e. the outer product of three vectors, this is just a trivector valued function. The following script gives an example of this. Figure 3 shows the visualization result of this script.

```
DefVarsN3(); // Define standard variables for conformal space
// Create a plane through points (0,0,1), (0,1,0), (0,0,-1) and 'e'
P = VecN3(0,0,1) ^ VecN3(0,1,0) ^ VecN3(0,0,-1) ^ e;

MyFunc =  // This is how a function is defined
{ // Function start
    a = _P[1]; // expect first parameter to be angle
    b = _P[2]; // expect second parameter to be radius
    c = _P[3]; // expect third parameter to be pos.

    S = SphereN3(0,0,c, b); // create a sphere
    C = S & P; // intersect sphere with plane P which gives a circle
    R = RotorN3(0,1,0, a); // create a rotor.

    // Now rotate circle
    (R * C * ~R) // No semicolon here since this is the return value
} // Function end

phi = 0;  // Initial angle value
Circle_list = MyFunc(phi, 1, 1); // first circle
Color_list = Color(1,0,0);       // first color
loop // start an infinite loop which needs to be stopped with 'break'
{
    if (phi > 2*Pi) // check the loop-end condition
        break;      // if satisfied, end loop.

    r = pow(cos(1.5*phi), 2) + 0.1; // make r a function of phi
    d = pow(cos(1*phi), 2) + 0.1;   // make d a function of phi

    Circle_list << MyFunc(phi, r, d); // add circle to list
    // Add corresponding color to list of colors
    Color_list << Color(r - 0.1, 0, 1.1 - r);

    phi = phi + Pi / 40; // Increase the value of phi
} // Jump to start of loop
```

```
:Circle_list;  // Visualize the list of circles
// Draw the surface over the circles using the corresponding colors
DrawCircleSurface(Circle_list, Color_list);
```

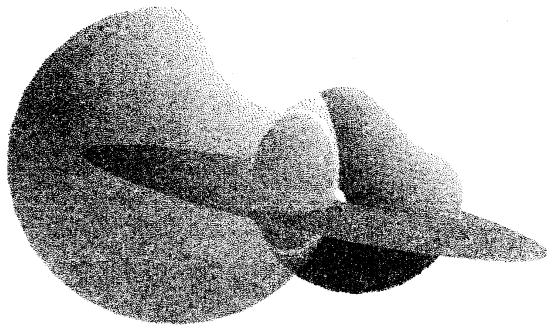

Figure 2: The meet of two spheres resulting in a circle. The meet of the circle with the plane results in a point pair.

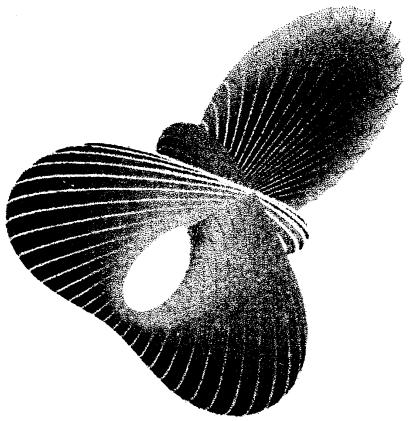

Figure 3: Surface swept out by a circle-valued function.

To make such surfaces look even better, it is also possible to add additional lights to the scene. Basically all lighting features offered by OpenGL can be accessed directly through CLU*Calc*. For details on how to use lights, please refer to [Per03].

2.1 User Interaction

Although static visualizations of GA formulas are already quite useful, some properties can be understood much better with interactive visualizations. To make a visualization user-interactive is quite easily done with CLU*Calc*. For example, to generate a vector that can be changed by the user, one may simply write `VecE3(1)`. The number passed to the function `VecE3` denotes a so called "mouse mode". This mouse mode can be set via a menu in CLU*Calc*. If, in this example, the user sets the mouse mode to 1, then holds down the right mouse button and moves the mouse, the return value of `VecE3(1)` is changed and the whole script is re-executed. Therefore, also anything that depends on the return value of `VecE3(1)` will be changed and re-displayed. For more details on mouse modes see [Per03]. The scripts we presented so far may thus very easily be made user-interactive, simply by replacing the initial `VecE3` functions from fixed values (e.g. `VecE3(1,0,0)`), to mouse modes (e.g. `VecE3(1)`).

Apart from generating user-interactive vectors, it is also possible to extract user-interactive scalar values which are related to certain mouse movements in given mouse

modes. The function which does this is called `Mouse`. For example, the return value of the function call `Mouse(1,2,1)` depends on the movement of the mouse along its x-axis, when mouse mode 1 is selected and the right mouse button pressed. The first parameter gives the mouse mode, the second one the mouse button (1: left, 2: right) and the third parameter gives the axis (1: x-axis, 2: y-axis, 3: z-axis). By default moving the mouse with the right mouse button pressed, changes the values for the x and z axes. When the "shift"-key is pressed at the same time, then the values for the x and y axes are changed. The `Mouse`-function for the left mouse button, i.e. second parameter set to 1, works very similar. The only difference is that the values returned lie in the range $[0, 2\pi[$. This is very useful when making rotors interactive. Here is an example script.

```
// Use mouse mode 1 and left mouse button
?a = Mouse(1,1,1);  // Value changed by movement in x-dir.
?b = Mouse(1,1,2);  // Value changed by pressing "shift"
                    // and moving in y-dir.
?c = Mouse(1,1,3);  // Value changed by NOT pressing "shift"
                    // and movement in y-dir.

:Red;                       // Set color to red
:Ry = RotorE3(0,1,0, a);    // Create rotor about y-axis with angle 'a'

:Color(0.1, 0.2, 0.8);      // Set color to given (r,g,b) values
:Rz = RotorE3(0,0,1, b);    // Create rotor about z-axis with angle 'b'

:Color(0.2, 0.8, 0.1);      // Set color to given (r,g,b) values
:Rx = RotorE3(1,0,0, c);    // Create rotor about x-axis with angle 'c'
```

The text output window of CLU*Calc* will now display the values of `x`, `y` and `z`, and the visualization window displays the rotors. Figure 4 shows an example visualization. A rotor is always visualized as a rotation axis with a partial disc perpendicular to it, representing the rotation plane and the rotation angle. By switching to mouse mode 1, holding down the left mouse button and moving the mouse, the visualization is changed continually, according to the mouse movement.

2.2 Animation

Apart from having user-interactive visualizations, it can be also very helpful to animate them. Both features can also be mixed, such that the user can interact with some animated features. Again, it is very simply to animate a script with CLU*Calc*. First CLU*Calc* has to be told that a script is to be animated. This is done with the script line `_DoAnimate = 1;`. Once the script is parsed, it is executed continually, with a maximum of 25 executions per second. The number of executions that can be achieved per second, depends on the script to be executed and the computer used.

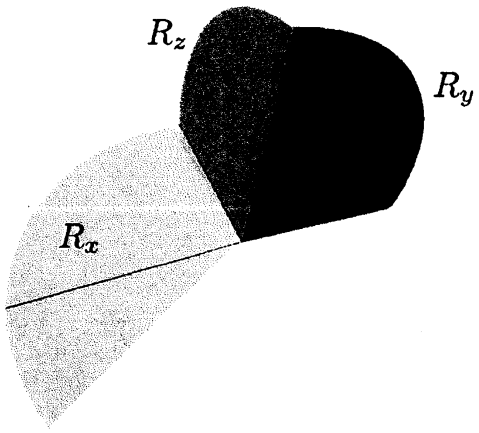

Figure 4: Three rotors about the x, y and z-axes.

In order to generate animated visualizations, two variables are now available: `Time` and `dTime`. While the first gives the time in seconds elapsed since the start of the animation, the latter gives the elapsed time since the last execution of the script. Even though the time variables give the time in seconds, their precision lies in the area of one millisecond. Here is an example script that uses animation.

```
// Enable animation of this script.
_DoAnimate = 1;

// Output time since start of animation in text output window.
?Time;

DegPerSec = 45; // Rotate with 45 degrees per second

// Evaluate current rotation angle. The variable 'RadPerDeg'
// is predefined and gives radians per degree. The variable
// 'Pi' is also predefined with the value of pi.
// The operator '%' evaluates the modulus.
?angle = (Time * DegPerSec * RadPerDeg) % (2*Pi);

:Black;
// This will become the user-interactive rotation axis with
// initial value (0,1,0), i.e. the y-axis.
:w = VecE3(1) + VecE3(0,1,0);

:Blue;
```

```
:R = RotorP3(w, angle); // The actual rotor about axis 'w'

:Red;
// The plane through points (1,0,0), (1,1,0), (1,0,1).
A = VecP3(1,0,0)^VecP3(1,1,0)^VecP3(1,0,1);

// The plane A rotated by R. Note that ~R denotes the reverse of R.
:B = R * A * ~R;
```

The animation feature may also be used to simulate physical effects like springs or gravitation. In this case it is usually necessary to initialize a set of variables during the first run of a script. Subsequent runs should then only adapt the variables' current values. This can be done in CLU*Calc* via the predefined variable **ExecMode**. The value of this variable may differ in each execution of a script and depends on the reason why the script was executed. To check for a particular execution mode, a bit-wise AND operation of **ExecMode** with one of a set of predefined variables has to be done. For example, if the script is executed because the user changed it, then **ExecMode & EM_CHANGE** is non-zero, where **&** is the operator for a bit-wise AND. This is used in the following example script, where a mass feels a gravitational pull towards the origin.

```
_DoAnimate = 1; // Make this script animated

// Check whether script has just been loaded
// or has been changed.
if (ExecMode & EM_CHANGE)
{
    // if true: intialize variables.
    TimeFactor = 0.01; // Factor for simulation timestep
    G = 6.67e-2; // Gravitational constant (in some units)

    pos = VecP3(0,1,0);    // Position vector in projective space
    vel = DirVecP3(6,0,0); // Direction vector in projective space

    Mass = 1; // Mass of object at origin
    O = VecP3(0,0,0); // the origin
}

?deltaT = dTime * TimeFactor; // Current time step

dist = abs(pos - O); // Distance of mass to origin
dir = (O - pos) / dist; // Normalized direction to origin
acc = G * Mass / (dist * dist); // Current acceleration
vel = vel + acc * dir; // Adapt velocity

:Red;
```

```
:pos = pos + vel * deltaT; // Adapt position and draw it
:Blue;
:0;
```

2.3 Annotating Graphics

It was mentioned above that it is possible to annotate graphics generated with CLU *Calc* using arbitrary LaTeX code. This is a very useful feature, because visualizations can become much more easy to understand when the different elements are labelled. The advantage of using LaTeX code is clearly that virtually all mathematical symbols are available. It is also convenient for producing illustrations for printed texts, since the same symbols can be used in the text and the illustration. In order for CLU*Calc* to be able to render LaTeX code, the LaTeX software environment and some other helper programs have to be installed. For the exact details see [Per03].

In CLU*Calc* LaTeX text can be rendered using the command `DrawLatex`. The text always has to be drawn relative to a point. For example, to draw the text "Hello World" at position $(1, 1, 1)$, one can write `DrawLatex(1,1,1, "Hello World")`. Instead of passing a string with the latex text, it is also possible to give a filename of a LaTeX file which is rendered instead. The text is rendered as a bitmap and then drawn with the bottom left corner at the position given. Since the text is drawn as a bitmap, it is not changed perspectively when moved around in the visualization space. It is also possible to give an arbitrary adjustment of the text bitmap relative to the point given (see help on `SetLatexAlign` in [Per03]).

The process of rendering a bitmap from LaTeX code is not particularly fast. Therefore, rendered text bitmaps are cached, so they only have to be rendered once when the script is loaded. However, this would mean that anybody who wants use a script that contains LaTeX annotations must have LaTeX installed. Furthermore, it can be quite annoying to always wait for a couple of seconds before all the LaTeX text for a script is rendered. For all of these reasons, the rendered LaTeX bitmaps can be stored in files and are then readily available when a script is loaded. For example, `DrawLatex(1,1,1, "Hello World", "text1")` renders the text "Hello World" and stores the rendered bitmap with a filename that is constructed from the name of the script and "text1". For more details on how to render LaTeX text see [Per03].

Apart from annotating geometric entities in a visualization, it is also practical to have text that does not move with the visualization, as for example a title for the visualization. This is also possible with CLU*Calc* by drawing text in an overlay. Note that anything, not just text, can be drawn in an overlay, which allows for a static fore- or background. The following script gives an example of this feature. The corresponding visualization is shown in figure 5.

```
DefVarsE3();              // Define variables for E3
```

```
// Set Latex magnification. This is chosen very high here, since this
// script was used to generate the corresponding figure for this text.
SetLatexMagStep(20);
SetPointSize(2);         // Size in which points are drawn
:DRAW_POINT_AS_SPHERE;  // Draw points not as dots but as small spheres
:E3_DRAW_VEC_AS_ARROW;  // Draw vectors in E3 as arrows.

:e1 :Red;        // Draw basis vector e1 in red.
:e2 :Green;      // Draw basis vector e2 in green.
:e3 :Blue;       // Draw basis vector e3 in blue.

:Black;          // Draw the following text in black
SetLatexAlign(0, 0.5); // Align text left/centered
DrawLatex(e1, "$\vec{e}_1$", "e1"); // Draw text

SetLatexAlign(0.5, 0); // Align text centered/bottom
DrawLatex(e2, "$\vec{e}_2$", "e2"); // Draw text

SetLatexAlign(1, 0.5); // Align text right/centered
DrawLatex(e3, "$\vec{e}_3$", "e3"); // Draw text

:E3_DRAW_VEC_AS_POINT;  // From now on draw vectors again as points
:Orange;         // Set color to orange
:A = VecE3(1);  // Draw user-interactive vector
:Black;                      // Set color to black for text drawing
SetLatexAlign(-0.05,0);      // Align text
DrawLatex(A,                 // Draw formula at position of 'A'
    "\[\oint_{\mathcal{S}}\, f(\vec{x})\, d\vec{x}\]",
    "formula");

StartOverlay(); // Now start an overlay to draw text that does not move
        // when the user rotates or translates the above visualization.
SetLatexMagStep(26);         // Increase size of Latex text rendering
SetLatexAlign(0,1);          // Align text left/top.
DrawLatex(5, 5, 0, "\sl The Title", "title");  // Draw Title of slide
EndOverlay();                // End overlay here
```

Together with the overlay feature, basically all prerequisites are available to give presentations with CLU*Calc*. Indeed, the other elements that are needed, like a full screen mode, stepping through a list of slides and using a white background, are also provided by CLU*Calc*. In the CLU*Calc* distribution, an example presentation is included, which can serve as a template for your own presentations. Of course, preparing presentations in CLU*Calc* is not as easy as using a tool like StarOffice or PowerPoint, but CLU*Calc* allows you to combine text with interactive and/or animated 3d-graphics. For more details on giving presentations with CLU*Calc*, please see [Per03].

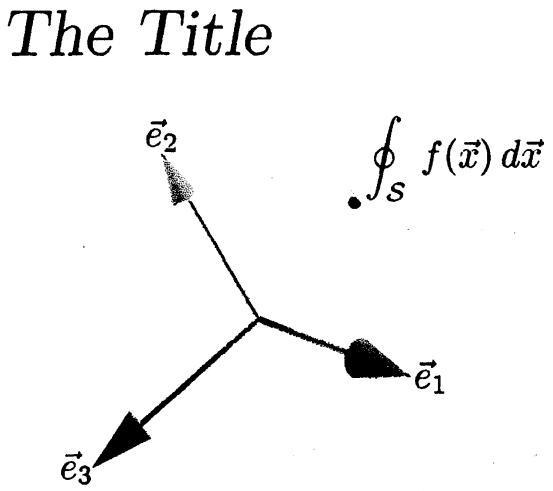

Figure 5: LaTeX text in a visualization.

2.4 Multivector Calculations

Apart from the visualization features, CLU*Calc* also supports advanced calculations with multivectors. A very useful operation is the inversion of a multivector, if the multivector has an inverse. The inversion is obtained with the ! operator. If a multivector has no inverse then zero is returned. For example, the script

```
DefVarsE3();              // Define variables for E3

?M = 1 + e1 + e2^e3;      // Define some multivector
?iM = !M;                 // Evaluate inverse of M
?"M * iM = " + M*iM;      // Check that iM is inverse of M

?W = 1 + e1;              // A non-invertible multivector
?iW = !W;                 // The inversion
```

results in the output

```
M = 1 + 1^e1 + 1^e23
iM = 0.2 + 0.2^e1 + -0.6^e23 + 0.4^I
M * iM = 1
W = 1 + 1^e1
iW = 0
```

The inversion of multivectors offers the possibility to solve simple multivector equations of the form $AX = B$ for X if A and B are known. More complex, linear equations can be solved by regarding multivectors as simple vectors in a higher dimensional vector space, and GA products as bilinear functions in this space. CLU*Calc* supports this way of looking at multivectors. The background to this functionality is the following.

The algebraic basis of a GA over a n-dimensional vector space has dimension 2^n. If we denote the algebraic basis of $\mathcal{C}\ell(\mathbb{R}^n)$, i.e. the GA over the vector space $\mathbb{R}^n$, by $\{E_i\}_{i=1}^{2^n}$, then an arbitrary multivector $A \in \mathcal{C}\ell(\mathbb{R}^n)$ can be written as

$$A = \sum_{i=1}^{2^n} \alpha^i E_i,$$

where the $\{\alpha^i\}_{i=1}^{2^n} \subset \mathbb{R}$ are scalars. The inner, outer and geometric product of GA between two multivectors may then be written as

$$A \circ B = \sum_{i=1}^{2^n} \sum_{j=1}^{2^n} \sum_{k=1}^{2^n} \alpha^i \beta^j g^k{}_{ij} E_k, \tag{1}$$

where $B = \sum_{i=1}^{2^n} \beta^i E_i$, $\circ$ is a placeholder for one of the products and $g^k{}_{ij}$ is a tensor whose entries are either 1, -1 or zero. This tensor encodes the particular properties of the product used. Therefore this tensor is different for each product. If a third multivector $C \in \mathcal{C}\ell(\mathbb{R}^n)$ is written as $C = \sum_{i=1}^{2^n} \gamma^i E_i$ and $C = A \circ B$, then the relation between the $\{\alpha^i\}$, $\{\beta^j\}$ and $\{\gamma^k\}$ follows from equation (1) as

$$\gamma^k = \sum_{i=1}^{2^n} \sum_{j=1}^{2^n} \alpha^i \beta^j g^k{}_{ij}. \tag{2}$$

Now suppose A and C are known and B is to be evaluated. This may be done by first evaluating the sum over i on the RHS, which then gives $\gamma^k = \sum_{j=1}^{2^n} \beta^j h^k{}_j$, where $h^k{}_j := \sum_{i=1}^{2^n} \alpha^i g^k{}_{ij}$. By writing the $\{\gamma^k\}$ and $\{\beta^j\}$ as column vectors c and b, respectively, and $h^k{}_j$ as matrix H, the above equation becomes $\mathsf{c} = \mathsf{H}\,\mathsf{b}$, which may be solved for b by inverting H. This is exactly what is done by CLU*Calc* internally, when the inverse of a multivector is evaluated.

If instead of A and C, in the above example, B and C are known and A is to be evaluated, then first the sum over j has to be evaluated. Taking care of the order of the indices i and j of $g^k{}_{ij}$ is important, since the products are in general not commutative, i.e. $A \circ B \neq B \circ A$ in general. In CLU*Calc* the function `GetMVProductMatrix` is used to evaluate the sum of the components of a multivector with the tensor $g^k{}_{ij}$ for a given product. The functions `MV2Matrix` and `Matrix2MV` transform multivectors to a vector representation and vice versa. Here is a simple example script that evaluates the geometric product of two multivectors using this vector representation.

```
DefVarsE3();  // Define variables for E3
A = e1;       // Define multivector A
B = e2;       // Define multivector B

// Evaluate the matrix for the product A * B of the
// components of A summed over index i of the
// tensor g^k_ij representing the geometric product.
Agp = GetMVProductMatrix(A, MVOP_GP, 1 /* from left  */);
// Do the same for product B * A with components of a
gpA = GetMVProductMatrix(A, MVOP_GP, 0 /* from right */);

// Transform multivector B in matrix representation.
// Bm is a column vector.
Bm = MV2Matrix(B);

// The operator '*' between two matrices evaluated
// the standard matrix product.
C1m = Agp * Bm; // Evaluate A * B
C2m = gpA * Bm; // Evaluate B * A

?C1 = Matrix2MV(C1m); // Transform C1m back to a multivector
?C2 = Matrix2MV(C2m); // Transform C2m back to a multivector
```

This script produced the text output

```
C1 = 1^e12
C2 = -1^e12
```

Many more things can be done with these functions. It is, for example, possible to map not a whole multivector to a vector, but only certain components. In this way, one can, for example, solve for a rotor numerically, while ensuring that the result can have only scalar and bivector components. For the details please refer to [Per03].

The matrices generated in the above script can also be visualized by CLU*Calc*. For this purpose the function `Latex` exists, which takes a matrix as parameter and returns the LaTeX code that draws the corresponding matrix. The following code snippet can be added to the end of the above script to draw the various matrices. The result of this drawing can be seen in figure 6. Not all details of this script can be explained here. Again [Per03] will be of help.

```
_BGColor = White;      // Set Background color to White
_FrameBoxSize = 0;     // Do not draw a frame box

StartOverlay();        // Start an overlay
```

```
SetLatexMagStep(14);  // Increase Latex font size
:Black;               // Set drawing color to black
SetLatexAlign(0,1);   // Set Latex alignment
// The DrawLatex function returns a list of values that
// give the extent of the rendered bitmap, so that more
// text can be drawn next to it.
P1 = DrawLatex(3, 5, 0,
    "\[ A\circ = " + Latex(Agp) + "\]", "mat");

P2 = DrawLatex(3, P1[2]+5, 0,
    "\[ \circ A = " + Latex(gpA) + "\]", "mat2");

DrawLatex(P1[4]+10, 5, 0, "\[ B = " + Latex(Bm) + "\]", "b");

DrawLatex(P2[4]+10, P2[5], 0, "\[ C_1 = " + Latex(C1m) + "\]", "c");
EndOverlay();
```

$$A\circ = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\circ A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad C_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Figure 6: Matrices drawn with LaTeX in the visualization window.

Another advanced feature of CLU*Calc* is the evaluation of products in GA with error propagation. This means that for each multivector a covariance matrix can be defined representing the uncertainty with which the multivector is known. It is then possible to evaluate the geometric, inner and outer product of multivectors with associated covariance matrices and propagate their uncertainties. Since intersections of geometric objects can in principle be evaluated using the inner, outer and/or geometric product, it is possible to do uncertain geometric reasoning with CLU*Calc*. Furthermore, since in conformal space circles and spheres are represented in a linear fashion, error propagation extends readily to these objects. One can, for example, evaluate the circle with associated covariance

matrix from three uncertain points, simply by evaluating their outer product with error propagation. Detail of how to use this functionality may again be found in [Per03].

3 Conclusions

The goal of this text was to give the reader a first impression of the features of CLU*Calc* and how they may be used for teaching geometry and subjects that profit from geometric visualizations. I hope that the versatility of CLU*Calc*, in terms of the different ways it may be used, became clear.

1. CLU*Calc* is a tool that analyzes GA entities with respect to their geometric meaning and visualizes them. This can be very useful when learning about the GAs of Euclidean, projective and conformal space, since students can play around with the different entities and *discover* what the inner, outer and geometric product mean geometrically.

2. CLU*Calc* can be used to write scripts that evaluate complex algorithms, and offers a platform to easily visualize results as interactive and/or animated 3d-graphics. This is certainly a useful aspect when doing research with GA.

3. CLU*Calc* allows the user to produce images for printed publications, that use the same LaTeX fonts as the printed text. By allowing for transparent surfaces and user defined lighting, the visual impact of 3d-graphics in a printed medium can also be largely increased.

4. All visualizations can be used right away in presentations. In particular, the combination of fixed text and interactive and animated 3d-graphics in a slide, furthers the understanding of complex geometric relations and is also bound to catch the attention of an audience.

The whole software package is released as OpenSource software, to give students a cost-free tool that supports their studies, but also in the hope that a number of researchers from the GA community will join the development of CLU*Calc* and thus make it a powerful platform that furthers the research in GA.

References

[DMB00] Dorst L., Mann S., Bouma T.: GABLE: A MatLab tutorial for geometric algebra. HTML document, 2000. Last visited 15. Sept. 2003, `http://carol.wins.uva.nl/~leo/GABLE/`.

[Dor01] DORST L.: Honing geometric algebra for its use in the computer sciences. In *Geometric Computing with Clifford Algebra* (2001), Sommer G., (Ed.), Springer-Verlag, pp. 127–151.

[GLD93] GULL S. F., LASENBY A. N., DORAN C. J. L.: Imaginary numbers are not real – the geometric algebra of space time. *Found. Phys. 23*, 9 (1993), 1175.

[GM91] GILBERT J. E., MURRAY M. A. M.: *Clifford algebras and Dirac operators in harmonic analysis.* Cambridge University Press, 1991.

[Hes86] HESTENES D.: *New Foundations for Classical Mechanics.* Dordrecht, 1986.

[HS84] HESTENES D., SOBCZYK G.: *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics.* Dordrecht, 1984.

[HZ91] HESTENES D., ZIEGLER R.: Projective Geometry with Clifford Algebra. *Acta Applicandae Mathematicae 23* (1991), 25–63.

[LFLD98] LASENBY J., FITZGERALD W. J., LASENBY A., DORAN C.: New geometric methods for computer vision: An application to structure and motion estimation. *International Journal of Computer Vision 3*, 26 (1998), 191–213.

[LHR01] LI H., HESTENES D., ROCKWOOD A.: Generalized homogeneous coordinates for computational geometry. In *Geometric Computing with Clifford Algebra* (2001), Sommer G., (Ed.), Springer-Verlag, pp. 27–59.

[Lou97] LOUNESTO P.: *Clifford Algebras and Spinors.* Cambridge University Press, 1997.

[Per00] PERWASS C.: *Applications of Geometric Algebra in Computer Vision.* PhD thesis, Cambridge Universtiy, 2000.

[Per03] PERWASS C.: The CLUScript online help. HTML document, 2003. Last visited 8. Feb. 2004, `http://www.perwass.de/CLU/CLUCalcDoc/`.

[PH03] PERWASS C., HILDENBRAND D.: *Aspects of Geometric Algebra in Euclidean, Projective and Conformal Space.* Technical Report Number 0310, CAU Kiel, Institut für Informatik und Praktische Mathematik, September 2003.

[PL01] PERWASS C., LASENBY J.: A Unified Description of Multiple View Geometry. In *Geometric Computing with Clifford Algebra* (2001), Sommer G., (Ed.), Springer-Verlag.

[Por95] PORTEOUS I. R.: *Clifford Algebras and the Classical Groups.* Cambridge University Press, 1995.

[Rie93] RIESZ M.: *Clifford Numbers and Spinors.* Kluwer Academic Publishers, 1993.